Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 13 – Initialization and Aggregates

Thomas Corbat / Felix Morgner
Rapperswil, 18/21.12.2023
HS2023

- **You can recognize and name different kinds of initialization**

- **You can explain the contraints imposed on aggregate types**

- **You can implement aggregate classes**

- **Recap Week 12**

- **Different Kinds of Initialization**

- **Aggregate Types**

Recap Week 12

- **Mix-in of functionality from empty base class**

  - Often with own class as template argument (CRTP) e.g., `boost::equality_comparable<T>`

  - No inherited data members, only added functionality

```cpp
struct Date : boost::equality_comparable<Date> {
  //...
};
```

- **Adapting concrete classes**

  - No additional own data members

  - Convenient for inheriting member functions and constructors

```cpp
template<typename T, typename Compare>
struct indexableSet : std::set<T, Compare> {
  //...
};
```

- **Implementing a design pattern with dynamic dispatch**

  - e.g., Strategy, Template Method, Composite, Decorator

  - Provide common interface for a variety of dynamically changing or different implementations

  - Exchange functionality at run-time

- **Base class/interface class provides a common abstraction that is used by clients**
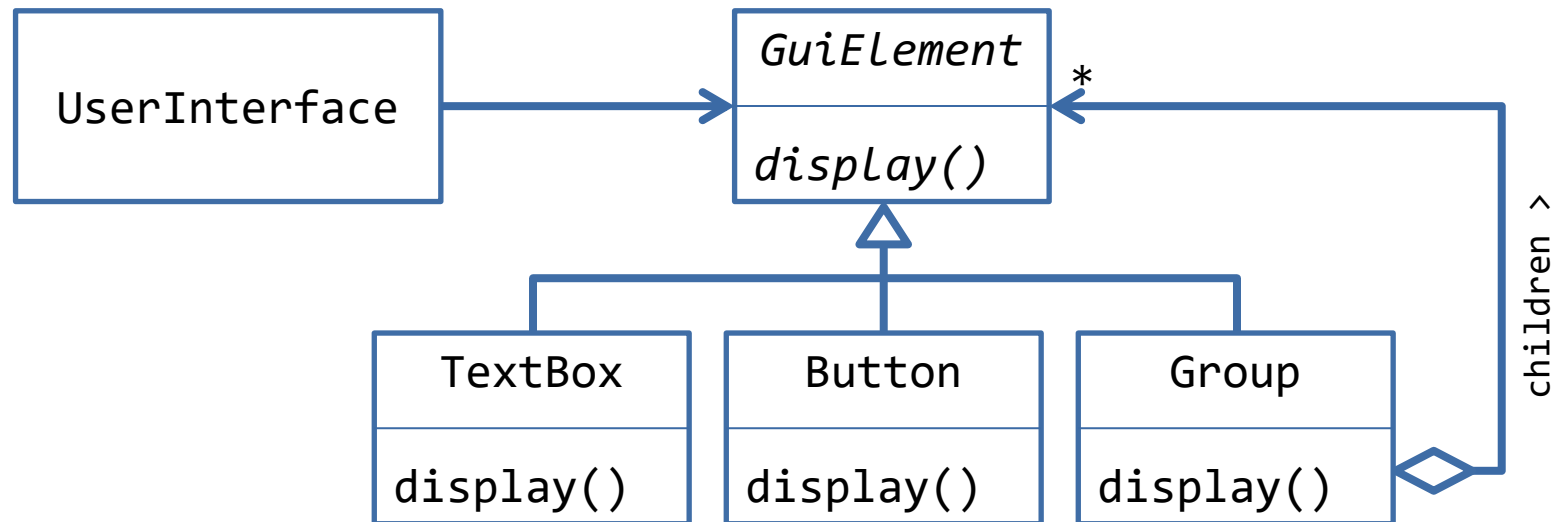
- **To override a virtual function in the base class the signature must be the same**

- **Constness of the member function belongs to the signature**

```cpp
struct Base {
  virtual void sayHello() const {
    std::cout << "Hi, I'm Base\n";
  }
};

struct Derived : Base {
  void sayHello() override {  ⚡
    std::cout << "Hi, I'm Derived\n";
  }
};

struct OtherDerived : Base {  ⚡
  void sayHello(std::string name) const override {
    std::cout << "Hi " << name << ", I'm OtherDerived\n";
  }
};
```

- **There are no interfaces in C++**

- **A pure virtual member function makes a class abstract**

- **To mark a virtual member function as pure virtual it has zero assigned after its signature**

  - = 0

  - No implementation needs to be provided for that function

```
struct AbstractBase {
    virtual void doitnow() = 0;
};
```

- **Abstract classes cannot be instantiated (like in Java)**

```
AbstractBase create() {
    return AbstractBase{};
}
```

- **You can declare the copy-operations as deleted**

```cpp
struct Book {
  //...
  Book & operator=(Book const & other) = delete;
  Book(Book const & other) = delete;
};

struct EBook : Book {
  //...
 EBook(EBook const & other) :
    Book{pages},
    currentPageNumber{other.currentPageNumber}{}
  EBook & operator=(EBook const & other) {
    pages = other.pages;
    currentPageNumber = other.currentPageNumber;
    return *this;
  }
};
```

```cpp
void readBook(Book book);

int main() {
  EBook designPatterns{"..."};
  readBook(designPatterns);

  EBook refactoring{"..."};
  Book & some = designPatterns;
  some = refactoring;
  EBook copy = designPatterns;
  copy = refactoring;
}
```

Kinds of Initialization

- **Default Initialization**

- **Value Initialization**

- **Direct Initialization**

- **Copy Initialization**

- **List Initialization**

- **Aggregate Initialization**

- **The kind depends on the context**

  - Four general syntaxes

1. **Nothing**

2. (`expression list`)

3. = `expression`

4. { `initializer list` }

- **Simplest for of initialization**

  - ◼ Simply don't provide an initializer

  - ◼ Effect depends on the kind of entity we declare

  - ◼ Does not work for references!

- **Danger lurks when using default initialized entities**

- **Does not necessarily work with `const`**

  - ◼ The object must have a "valid" value

```cpp
int global_variable; // implicitly static

void di_function() {
  static long local_static;

  long local_variable;
}

struct di_class {
  di_class() = default;

  char member_variable; // not in ctor init list
};
```

- **Static variables are**

  - zero initialized first,

  - then their type's default constructor is called

```cpp
int global_variable; // implicitly static

std::string global_text;

void di_function() {
  static long local_static;
}
```

- **If the type cannot be default constructed, the program is ill-formed!**

```
struct blob {
    blob(int);
};

blob static_instance;
```

Suppresses Default Constructor

```
error: no matching function for call to 'blob::blob()'
```

- **Non static integral and floating point variables are uninitialized**

- **Objects of class types are constructed using their default constructor**

- **Member variables not in a ctor-init-list are default initialized**

- **Arrays initialize all their elements accordingly**

```cpp
auto di_function() -> void {
  long local_variable;
  std::string local_text;
}

struct di_class {
  di_class() = default;
  char member_variable; // not in ctor init list
};
```

- **Danger lurks!**

  - ■ Reading an uninitialized value incurs undefined behavior!

```cpp
auto print_uninitialized() -> void {
  int my_number;
  std::cout << my_number << '\n';
}
```

DANGER

Undefined Behavior

- **Initialization performed with empty ( ) or { }**

  - { } is preferrable, since it works in more cases

- **Invokes the default constructor for class types**

```cpp
#include <string>
#include <vector>

auto vi_function() -> void {
  int number{};
  std::vector<int> data{};
  std::string actually_a_function();
}
```

- **Similar to Value Initialization**

  - Uses non-empty ( ) or { }

  - When using { } only applies if not a class type (see List Initialization)

- **"Most vexing parse" lurks with ( )**

  - Prefer { ... }

```cpp
#include <string>

auto diri_function() -> void {
  int number{32};
  std::string text("CP1");
  word vexing (std::string());
}
```

- **Two interpretations**

  - Initialization with a value-initialized string

  - Declaration of a function returning word and taking an unnamed pointer to a function returning a string

- **The first is what we would expect**

- **The second is the one the standard requires!**

  - Therefore, prefer { … }

```
word vexing (std::string());
```

- **Initialization using =**

  - If the object has class type and the right-hand side has the same type

    - If the right-hand side is a temporary, the object is constructed "in-place"

    - Otherwise, the copy constructor is invoked

  - Otherwise, a suitable conversion sequence is searched for

- **Also applies to return statements and throw/catch**

```cpp
#include <string>

auto string_factory() -> std::string { return ""; }

auto ci_function() -> void {
  std::string in_place = string_factory();
  std::string copy = in_place;
  std::string converted = "CPl";
}
```

- **Uses non-empty { }**

  - Direct List Initialization

    ```
    std::string direct{"CP1"};
    ```

  - Copy List Initialization

    ```
    std::string copy = {"CP1A"};
    ```

- **Constructors are selected in two phases**

  - If there is a suitable constructor taking `std::initializer_list`, it is selected

  - Otherwise, a suitable constructor is searched

- **Since the std::initializer_list constructor is preferred, you might run into trouble**

```cpp
// vector(size_type count,
//        const T& value,
//        const Allocator& alloc = Allocator());

auto ouch() -> int {
  std::vector<int> data{10, 42};
  return data[5];
}
```

DANGER

Undefined Behavior

Aggregate Types

- **Simple class types**

  - Can have other types as public base classes

  - Can have member variables and functions

  - Must not have virtual member functions

  - Must not have user-declared or inherited constructors

  - Must not have protected or private direct non-static data members

- **Mostly used for "simple" types**

  - No invariant that must be established

  - Example: DTOs

- **Arrays are also Aggregates**

```cpp
struct person {
  std::string name;
  int age{42};

  auto operator<(person const & other) const -> bool {
    return age < other.age;
  }

  auto write(std::ostream & out) const -> void {
    out << name << ": " << age << '\n';
  }
};


auto main() -> int {
  person rudolf{"Rudolf", 32};
  rudolf.write(std::cout);
}
```

```cpp
struct db_entry {};

struct person : private db_entry {
  std::string name;
  int age{42};

  person() = default;

  auto operator<(person const & other) const -> bool {
    return age < other.age;
  }

  virtual auto write(std::ostream & out) const -> void {
    out << name << ": " << age << '\n';
  }
};
```

- **Special case of List Initialization**

  - If the type is an aggregate, the members and base classes are initialized from the initializers in the list

- **If more elements than members (or bases) are given the program is ill-formed**

- **Can also provide less initializers than there are bases and members:**

  - If the "uninitialized" members have a member initializer, it is used

  - Otherwise, they are initialized from empty lists

```
person rudolf{"Rudolf"};
```

Age will be set to 42

- **Numerous different kinds of initialization**

  - Avoid default initialization because of possible UB

  - Generally, prefer initialization with { }

  - Use ( ) only when aiming for a certain constructor (avoiding std::initializer_list constructors)

- **Aggregates can reduce code for simple types**

  - Only use them if you type has no invariant